ゲームプログラミング

DirectX

第9回

DirectPlayによるネットワークゲーム

恋塚昭彦，花田秀次(Bio_100%)

今回はDirectPlayを使ったネットワーク対応ゲームの開発について取りあげます。ゲームのネットワーク対応は，対戦が可能なゲームではもはや常識になっています。

ネットワーク対戦

家庭用ゲーム機では2台を接続して対戦できるネットワーク対戦ゲームのタイトルが増えており，またパソコンでは多人数同時協力/対戦プレイのできるゲームがLANやモデムを介した環境でとくに海外タイトルが早くからサポートしています。いまはインターネットの普及の追い風を受けて，商用ゲームサーバサービスなども現れ，それをサポートするゲームも出てきています。

とくにレースゲームやDOOMタイプのゲームにおいて，ネットワーク対戦機能がサポートされるのには簡単な理由が存在します。人間同士の対戦プレイは，コンピュータ相手よりも格段におもしろいからです。また，プレイヤーがゲーム中で取る多様な動作をすべて予測して実装することなど不可能でしょうし，学習機能を入れてもコンピュータが人間の脳より賢く，また適度に間抜けな存在になるにはまだまだ時間がかかりそうです。このジャンルでは通信機能は必須といえるでしょう。

以前は，パソコンでネットワークに対応させたゲームの作成は，たいへんな労力がかかりました。一口にネットワークといっても，さまざまなネットワークプロトコルが存在します。モデムのサポートやTCPのサポートなど，それぞれのプロトコルに対応させるだけでも骨が折れる作業でした。しかし，いまはDirectPlayがプロトコルの差異を吸収し，インターネットやモデム接続，シリアルケーブル直結まで対応できるのです。

DirectPlayの登場でプロトコル実装の煩雑さが取り除かれたといっても，ネットワークゲームを作るにはまだ考慮しなければならない点があります。DirectPlayの具体的な使用方法の前に，ネットワークゲームの設計について整理しておきます。

ネットワークゲームの設計

クライアント数

まずは自分の作りたいゲームが，何台のクライアントを必要とするのかを決定しましょう。2人対戦だから2台とすれば，苦労はかなり減ります。初めて通信対戦を実装するならば，まず2台のみの構成を薦めます。2台ならばネットワーク全体でのトラフィック(通信量)も少なく，接続形態もシリアルケーブル直結やモデム接続といった，比較的デバッグしやすい環境で実現可能だからです(Fig. 1)。3台以上ではサーバが必要となることもあります。クライアントのコーディング面では2台と3台以上ではそれほど労力の差はありませんが，システムのパフォーマンス向上にかかる

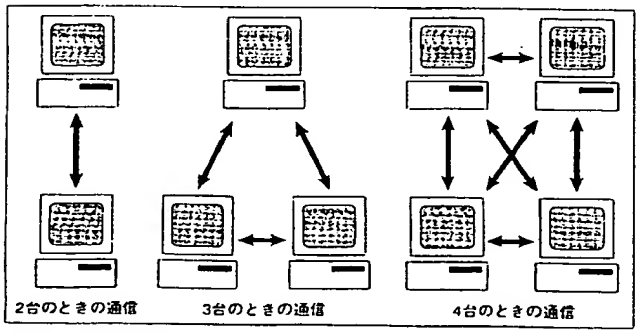


Fig. 1 トラフィック

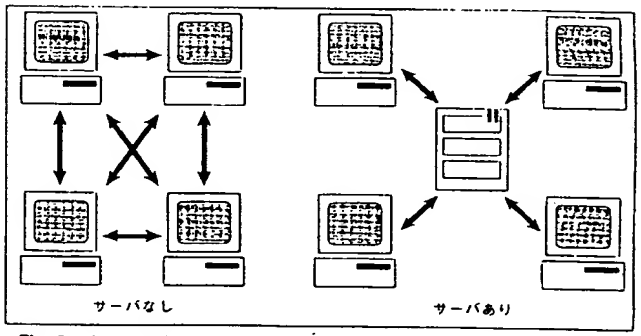


Fig. 2 サーバの有無

調整で費やされる時間に大きな違いが出てくるのです。

サーバが必要となるのは，クライアント数が多く，システム全体のトラフィック削減のためにサーバですべてのクライアント情報を集めて同報配信する場合です。サーバがなければ各クライアントから自分以外のすべてのクライアントに情報を送る必要が発生するので，ネットワーク全体のパフォーマンスが下がることがあります(Fig. 2)。いずれにしても3台以上の環境は，LANなどの潤沢な環境があるなら別ですが，個人ベースでは開発もプレイも難があります。

通信内容の最適化

次に，通信内容を考えてみましょう。ゲーム内で扱っているすべての情報をほかのクライアントに伝えることができれば，それにこしたことはありません。しかし，ゲームに登場するオブジェクトの情報が刻々と変化するリアルタイムゲームにおいて，そんなに多くのデータを流せないのが現状です。

たとえば28,800bpsの通信速度のインフラを使用して通信対戦すると秒間に3,600バイト送れます。仮に1/20秒刻みで情報を更新し続けると1回の更新は180バイトです。お互いにそれぞれのデータを送り合うとすれば90バイトで，そのうえプロトコルのヘッダ部分に必要なバイト数も差し引かれます。その範囲でやり取りしなければならないので，本当に必要な情報だけを厳選する必要があります。

レースゲームなら，最低限必要な情報は自分の車の位置とベクトルということになると思いますが，それ以外にいわゆるCOM車がいた場合はどうなるでしょう？　これはCOM車の動きのアルゴリズムに依存します。COM車の動きの決定に乱数を使用すれば，各パソコンごとにCOM車が違った動きをすることは確実です。これではいけません。回避方法として，自車と同じようにCOM車の位置やベクトルを交換し合って同期を保つことも考えられますが，ここはランダムな要素をアルゴリズムから排除して，とくに毎回位置情報を更新しなくてもそれぞれのパソコンで同じ動きをするようにして，通信しなければならないデータ量を減らすことを優先すべきでしょう。ランダムな要素がゲームに欠かせない場合は，乱数の系列と種を初期段階に合わせて，それぞれ乱数を発生させて同期を取ることも可能でしょう。同期しなければならない最低限必要なデータを追求すると，ユーザからのキー入力(あるいはジョイスティックの入力)の情報だけでよいことになります。

通信障害の回復

ゲームにかぎらずプロトコルを設計する際には，通信中にデータが相手に届かないことを考慮することが必要です。DirectPlayも送信したデータが必ず相手に届くことを保証するものではないことがドキュメントに記されています。多くのデータがボロボロと失われるのは，よほど質の悪いコネクションを確立した状況下でなければ起きませんが，データが失われる事象があることを前提条件としてプロトコルを開発する必要があります。とくにキー入力情報のみを通信相手とやりとりする場合は，通信データ量は少なくなりますが，データを失った際の回復方法の設計をしっかりしておかなければなりません。

通信内容を，位置の情報などといったある程度アプリケーションに意味のある情報にすると，データを失った際の回復のための実装が比較的簡単な場合があります。受信データが抜けても，常に最新の受信データに基づいて更新すればいいだけの場合もあるからです。キー入力情報のみの場合は，受信漏れをしたあとに新しいキー入力情報を受信したとしても，間の入力が抜けてヘンな操作を再現してしまいます。

エラーの回復方法のもっとも基本的な方法は，1回に送り合うパケット(送受信データ)に連番を持たせ，受信側が連番でデータを受信しているかをチェックするのです。もし番号が連続していなければ，その間のパケットを再度送ってもらいます(Fig. 3)。エラーが多発するようなら，ユーザに通信品質が悪いことを伝え，ゲームをきちんと強制終了させる仕組みも必要です。

画面との同期

アーケードゲーム機などでは，画面の更新間隔とデータ同期の間隔は一致しているようです。しかしパソコンゲームのユーザレベルではLAN環境も普通ないでしょうし，モデム接続による対戦やモデムからインターネットを通して対戦といった通信速度の遅い

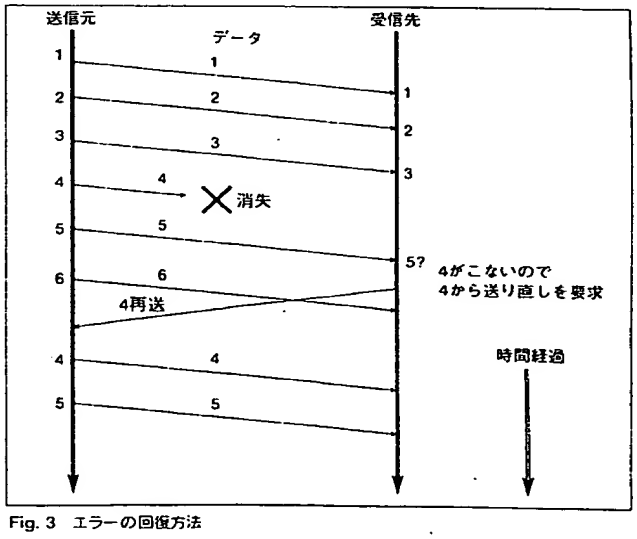


Fig. 3　エラーの回復方法

環境しかないのが現状です。スループットの不安定なインターネットを介されたら，1/30フレーム毎秒の更新間隔に合わせてデータの同期を行うことは現状ではとても不可能で，毎秒1回でも難しいでしょう。

しかし世の中には，インターネットを介した対戦をサポートしたゲームが多くあります。それらは，データの同期間隔を描画間隔と合わせずに何とかアクションゲームとして見せかけているのがほとんどです。画面更新間隔はパソコンのパフォーマンスをフルに引き出す一方，通信によるデータの同期は通信インフラのスループットに合わせています。プレイヤーは自分のキャラクタ(や車)はスムースに動かせますが，対戦相手の動きは飛び飛びになるような感じです。ゲームによっては他プレイヤーの動きをベクトルで持ち，通信が途切れている間の動きを予測して補完しているものもあります。こうしたやり方は現実に即したものですが，その反面イベントの同期がいっそう難しくなります。通信速度の遅いインフラでプレイすると，突然撃たれて死ぬように見えることも起きます。その点で限界はありますが，手間さえ惜しまなければマシンのパフォーマンスを通信部分に引きずられないようにするという意味で，画面更新間隔とデータ同期の間隔を別にするメリットが十分あります。3Dゲームでは，キー入力間隔と画面更新間隔が別なのは常識なので，比較的導入しやすい手法です。

モデムでの注意

モデムの使用をユーザに薦める場合，通信速度の設定には注意したほうがいいでしょう。一般にいわれるモデム速度は，データを絶え間なく流した際に達成する最大パフォーマンスです。小さなデータを細切れに流した場合，通信速度が速い設定がもっともパフォーマンスを引き出せるとはかぎりません。それはモデムによるレイテンシのために起こります。

一般に，レイテンシは高い通信速度を達成するプロトコルほど大きくなります。つまり最高通信速度28,800bpsのモデムでも，通信内容によっては14,400bpsや9,600bpsで通信したほうが速い場合が多々あります。家庭用ゲーム機用の通信アダプタがさほど高い通信速度に設定されていないのは，レイテンシを重視したためでしょう。モデム対戦サポートのゲームを開発したら，実際にどの通信速度がもっともパフォーマンスを出せるかを試して，それを設定することをユーザに推奨したほうがいいでしょう。

DirectPlayによる実装

DirectPlayはほかのDirectXコンポーネントとは異なり，ハードウェアを直接制御するものではありません。DirectPlayは，リアルタイム対戦型ゲームのための通信インタフェイスを提供します。最大のメリットは，開発者は下位のプロトコル(DirectPlayではサービスプロバイダと呼ぶが，ここではプロトコルと呼ぶ)に依存しないネットワーク対応コードを書くことができるところにあります。DirectPlayは，以下の通信プロトコルに対応しています。

・モデム接続
・LANにおけるTCPコネクション
・LANにおけるIPXコネクション
・シリアルケーブル接続
・インターネットプロバイダを介したTCPコネクション

プレイヤーは環境に応じてプロトコルを選択してゲームをプレイしますが，アプリケーション側でどのプロトコルを使用しているかを意識する必要がありません。バンド幅のみを考慮すればいいのです。またアプリケーションは，DirectPlayオブジェクトにバンド幅やレイテンシを問い合わせることや，過去のプレイ時のパフォーマンス情報の照会が可能なので，常にネットワークインフラに適合したゲームの進行管理を行えます。またプロトコルがゲーム用に限定して抽象化されているので，将来の新しいプロトコルも，DirectPlayが対応さえすればアプリケーション側はコードの変更をほとんどすることなくその恩恵を受けられるでしょう。実際にDirectX 3では，使用できるプロトコルスタックが増えています。

DirectPlayはほかのDirectXコンポーネントと同じく，COMを使って実装されています。この一連のインタフェイスを引き出すために，5個のAPIが用意されています(Fig. 4)。DirectPlayCreate APIは，特定のサービスプロバイダへのDirectPlayオブジェクトのインスタンス生成に使用します。システムに組み込まれたサービスプロバイダ(DirectPlayサーバ)は，DirectPlayEnumcrate APIにより取得できます。通常DirectPlayEnumerate APIにより利用可能なサーバのリストを取得してユーザに提示され，ユーザに選択されたサービスプロバイダのIDを引数としてDirectPlayCreateを実行します(Table 1)。

それぞれのゲームは，固有のGUIDでネットワークインフラ内で識別されます。GUIDは，ネットワークカードを持つ任意のPC上で，Win32SDKで提供されるuuidgen.excによって生成できます。ほかのコンピュータとのコネクションを確立するには，Direct Play Openメンバを実行します。一度通信コネクションの確立が成功すると，ゲームアプリケーションは個々の参加者に対して

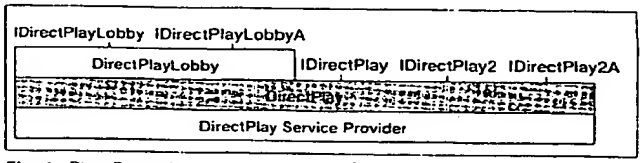


Fig. 4 DirectPlayのAPI

DirectPlayオブジェクトを使用するプレイヤーを作成します。ゲームアプリケーションは、プレイヤーIDを使用して任意のプレイヤーとメッセージを送受信可能です。ゲームセッション(ゲームのインスタンス)内のプレイヤーが抜けたり、新しいプレイヤーが加わった場合、おのおののプレイヤーはそのプレイヤーの名称、およびそのとき起きた状況変化についてのシステムメッセージを受け取ります。

DirectPlayは、それ自体が同期式アプローチをとっていません。ゲームアプリケーションのアーキテクチャに対して制限を加えないようにするためです。

DirectPlayの使用方法

DirectPlayのコネクションの確立から終了まで、ゲームとしての一般的手順を具体的なコード例をあわせて解説します。

①サービスプロバイダの検出

DirectPlayEnumerateを呼び出し、サービスプロバイダのリストをコールバック関数で取得しつつ作成します。そのあと、ユーザに対して使用すべきサービスプロバイダを、返されたサービスプロバイダの1つから選択するように要求します(List 1)。

②DirectPlayオブジェクトの作成

選択されたプロトコルを引数としてDirectPlayCreateを実行し、IDirectPlayインタフェイスを作成し、そのQueryInterfaceを通してIDirectPlay2Aインタフェイスを得ます(List 2)。

③ゲームセッションの探索

IDirectPlay2A::EnumSessionsメンバを使用して、ユーザの要

Table 1 DirectPlay API

HRESULT DirectPlayEnumerate(LPDPENUMCALLBACK lpCallback, LPVOID lpContext)		
内　容	システムにインストールされたサービスプロバイダを列挙する	
引　数	LpCallback：システムにインストールされたそれぞれのDirectPlayサービスプロバイダの情報から、呼び出される以下のインタフェイス仕様を持ったユーザのコールバック関数を指すポインタ	
	BOOL WINAPI EnumDPCallback(LPGUID lpguidSP, LPSTR/LPWSTR lpSPName, DWORD dwMajorVersion, DWORD dwMinorVersion, LPVOID lpContext)	
	引　数	LpguidSP　：DirectPlayサービスプロバイダのGUID
		LpSPName　：ドライバの記述を含んだ文字列を指すポインタ型はUnicodeを使うかどうかで異なる(Unicodeの場合はLPWSTR)
		DwMajorVersion：ドライバのメジャーバージョン番号
		DwMinorVersion：ドライバのマイナーバージョン番号
		LpContext　：呼び出し側で定義されるコンテキストを指すポインタ
	戻り値	TRUE　：列挙を続ける FALSE：列挙を止める
	LpContext　：列挙型コールバックを実行する際に毎回渡される呼び出し側で定義されるコンテキストを指すポインタ	
返り値	DP_OK　：OK DDERR_GENERIC　：未定義のエラー DPERR_EXCEPTION　：例外エラー DPERR_INVALIDPARAMS：パラメータ不正	
HRESULT DirectPlayCreate(LPGUID lpGUID, LPDIRECTPLAY FAR *lplpDP, IUnknown FAR *pUnkOuter)		
内　容	DirectPlayオブジェクトのインスタンスを生成する	
引　数	LpGUID　：生成されるべきドライバを示すGUIDを指すポインタ PlpDP　：作成したインタフェイスへのアドレスを得るポインタ変数のアドレス PunkOuter：予約(NULLを指定)	
返り値	DP_OK　：OK CLASS_E_NOAGGREGATION：pUnkOuterにNULL以外が入れられた DPERR_EXCEPTION　：例外エラー DPERR_UNAVAILABLE　：このバージョンではできません DPERR_INVALIDPARAMS　：パラメータ不正	

求に合った既存のゲームセッションを探索します。ユーザがすでにあるゲームセッションに参加するのではなく、新しいゲームセッションの作成を決定している場合は、探索する必要がありません。EnumSessionsは、通常DirectPlayオブジェクトが生成された直後に実行します。セッションへの接続中やゲームがセッションを生成したあとは実行できません。DirectPlayの返答への待ち時間の合計はdwTimeoutパラメータでコントロールされますが、0を設定することでプロトコルごとのデフォルトの値が採用されます。待ち時間を過ぎれば、DPESC_TIMEDOUTフラグがコールバックに通知され、lpDPSGameDescにNULLが入ります。そこで*lpdwTimeOutに新しい値を設定してTRUEを返せば探索を延長できます。FALSEを返せば探索を中止できます(List 3)。

④既存のゲームセッションへ参加する場合

ユーザが既存のゲームセッションから参加すべきセッションを選択したあと、コネクション初期化のためにIDirectPlay2::Openを実行します。このメンバはコネクションを実際に確立します。モデムコネクションの場合は実際に電話をかけます。既存のセッションへの参加の場合は、DPOPEN_JOINフラグを使用し、EnumSessionsで獲得したセッションIDを指定します(List 4)。

⑤新しいゲームセッションを作成する場合

新しいセッションを作成する場合、セッションはDPOPEN_CREATEフラグを使用してIDirectPlay2::Openで作成されます(List 5)。

⑥ローカルプレイヤーID作成

ローカルプレイヤーID作成のためにCreatePlayerを呼びます。IDirectPlay2::CreatePlayerは、現在のゲームセッションでのプレイヤーを生成する際に使用されます。返却されるプレイヤーIDは、プレイヤーのメッセージの行き先を指定したりプレイヤーを管理するために内部で使用されます。またアプリケーションは、IDirectPlay2::SetPlayerNameとIDirectPlay2::SetPlayerDataで任

List 1

1 サービスプロバイダの検出

```
DirectPlayEnumerate( DirectPlayEnumerateCallback, (LPVOID)hWndCtl );

// DirectPlayEnumerateの為のコールバック関数
BOOL FAR PASCAL DirectPlayEnumerateCallback(
    LPGUID lpSPGuid, LPTSTR lpszSPName, DWORD dwMajorVersion,
    DWORD dwMinorVersion, LPVOID lpContext)
{
    // サービスプロバイダ(プロトコル)名:lpszSPNameとlpSPGuidを
    // ダイアログボックス等のリストに追加しています。
}
```

List 2

2 DirectPlayオブジェクトの作成

```
HRESULT CreateDirectPlayInterface(LPGUID lpguidServiceProvider,
    LPDIRECTPLAY2A *lplpDPlay2A)
{
    LPDIRECTPLAY  lpDPlay1 = NULL;
    LPDIRECTPLAY2A lpDPlay2A = NULL;
    HRESULT       hr;

    lpDPlay1 = NULL;
    lpDPlay2A = NULL;
    // 従来のDirectPlayインターフェイスを作成します
    hr = DirectPlayCreate(lpguidServiceProvider, &lpDPlay1, NULL);
    if (hr != DP_OK)
        goto FAILURE;

    // クエリーを通してANSI版DirectPlay2インターフェイスを得ます。
    hr = lpDPlay1->QueryInterface(IID_IDirectPlay2A,
        (LPVOID *) &lpDPlay2A);

    if (hr != DP_OK)
        goto FAILURE;

    *lplpDPlay2A = lpDPlay2A;

FAILURE:
    if (lpDPlay1)
        lpDPlay1->Release();

    return (hr);
}
```

List 3

3 ゲームセッションの探索

```
// EnumSessionsを呼ぶ
hr = lpDPlay2A->EnumSessions(&sessionDesc, 0, EnumSessionsCallback,
    hWnd, DPENUMSESSIONS_AVAILABLE);

// EnumSessionの為のコールバック関数
BOOL FAR PASCAL EnumSessionsCallback(
    LPCDPSESSIONDESC2 lpSessionDesc, LPDWORD lpdwTimeOut,
    DWORD dwFlags, LPVOID lpContext)
{
    HWND hWnd = (HWND) lpContext;

    if ( dwFlags & DPESC_TIMEDOUT ) {
        return FALSE;          // これ以上セッションを探索しない
    }

    // セッション名lpSessionDesc->lpszSessionNameA
    // ダイアログボックス等のリストに追加していきます。

    return(TRUE);
}
```

List 4

4 既存のゲームセッションへ参加

```
DPSESSIONDESC2 sessionDesc2;

memset(&sessionDesc, 0x00, sizeof(DPSESSIONDESC2));
sessionDesc.dwSize = sizeof(sessionDesc);
sessionDesc.dwSize = sizeof(DPSESSIONDESC2);
sessionDesc.guidInstance = *lpguidSessionInstance;

hr = lpDPlay2A->Open(&sessionDesc, DPOPEN_JOIN);
if (hr != DP_OK)
    return FALSE; // セッション参加失敗
```

List 5

5 新しいゲームセッションを作成

```
DPSESSIONDESC2 sessionDesc;

memset(&sessionDesc, 0x00, sizeof(DPSESSIONDESC2));
sessionDesc.dwSize = sizeof( sessionDesc);
sessionDesc.dwFlags = DPSESSION_MIGRATEHOST | DPSESSION_KEEPALIVE;
sessionDesc.guidApplication = DPCHAT_GUID;
sessionDesc.dwMaxPlayers = MAXPLAYERS;
sessionDesc.lpszSessionNameA = lpszSessionName;

hr = lpDPlay2A->Open(&sessionDesc, DPOPEN_CREATE);
if( hr != DP_OK )
    return FALSE; // セッション作成失敗
```

意の名前とデータを割り付けることができます。これらの名前とデータは内部では使用されず，一意である必要もありません。DirectPlayで割り当てられるプレイヤーIDはセッション内では常に一意です。

成功に実行された場合，この関数は新しいプレイヤーがセッションに加入したことを告示するDPSYS_CREATEPLAYERORGROUPシステムメッセージをゲームセッションのほかのすべてのプレイヤーに送付します(List 6)。

⑦メッセージの送受信

さてDirectPlayでの必要な前準備は終わりました。ここから自由に送受信が可能となるわけです。

ゲームは，自分自身へのメッセージはIDirectPlay2A::Receiveメンバで受信します。発信プレイヤーIDがDPID_SYSMSG以外の場合はほかのプレイヤーからのメッセージ，発信プレイヤーIDがDPID_SYSMSGの場合はホストからのシステムメッセージです(List 7)。

メッセージを受信した際には，fromIDが0かどうかを判定してシステムメッセージであるかどうかを常に検知しなければなりません。DPMSG_ADDPLAYERメッセージは新たなるプレイヤーの参加を通知するメッセージなのでとくに重要です。受信バッファの先頭には，メッセージID(Table 2)が格納されています。

List 6

6 ローカルプレイヤーID作成

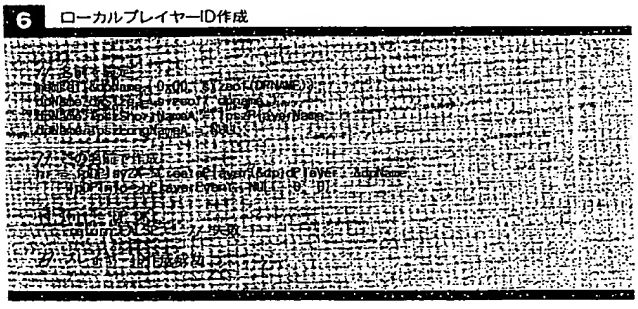


Table 2 メッセージID一覧

ID	内容
DPMSG_DESTROYPLAYERORGROUP	プレイヤーかグループを削除
DPMSG_CREATEPLAYERORGROUP	プレイヤーかグループを作成
DPMSG_ADDPLAYERTOGROUP	グループにプレイヤー参加
DPMSG_DELETEPLAYERFROMGROUP	グループからプレイヤーを削除
DPMSG_SETPLAYERORGROUPNAME	プレイヤー(グループ)名を設定
DPMSG_SETPLAYERORGROUPDATA	プレイヤー(グループ)データを設定
DPMSG_HOST	セッションのホストを委譲
DPMSG_SESSIONLOST	セッション切断

IDirectPlay2A::Sendメンバは，ほかのプレイヤー，ほかのプレイヤーのグループ，セッション内の全プレイヤーにメッセージを送ることができます。ほかのプレイヤーにメッセージを送るには，単に送りたい相手のプレイヤーIDを指定するだけです。プレイヤーのグループにメッセージを送るには，グループ生成で割り当てられたプレイヤーIDにメッセージを送ります。全セッションにメッセージを送るには，受信プレイヤーIDをDPID_ALLPLAYERSとしてメッセージを送ります。

IDirectPlay2::Sendは，IDirectDrawSurface2::LockからIDirectDrawSurface2::Unlockの間，またはIDirectDrawSurface2::GetDCからIDirectDrawSurface2::ReleaseDCの間で使ってはいけません(List 8)。

⑧ゲームセッションの終了

Closeメンバは，すでにオープンされているコネクションを閉じます。ローカルに作成されたすべてのプレイヤーは，ほかのセッション参加者に対してDPMSG_DESTROYPLAYERORGROUPシステムメッセージを送付して消去されます。

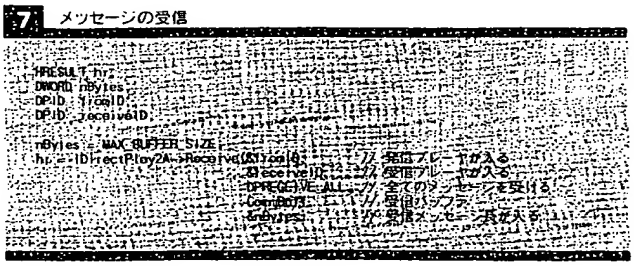
```
IDirectPlay2->Close();
```

今後

今回は，ネットワークゲーム設計の指針とDirectX 3でアップデートされたDirectPlay2.0の基本的な使い方で終わってしまったので，今後の連載でサンプルコードを含んだ解説記事を行う予定です。本連載でのご意見ご感想は，http://bio.and.or.jp/cgi-bin2/directbbs.cgiでお待ちしています。

List 7

7 メッセージの受信



List 8

8 メッセージの送信